

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

КУРСОВАЯ РАБОТА
по дисциплине «Алгоритмы и Структуры Данных»
Тема: Сортировки. Демонстрация

Студент гр. 8304

Птухов Д.А.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2019

ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ

Студент Птухов Денис Александрович

Группа 8304

Тема работы: сортировки

Исходные данные:

Написать программу для реализации сортировки слиянием (итеративным и рекурсивным подходом) и быстрой сортировки (итеративным и рекурсивным подходом), файловое считывание и файловая запись.

Содержание пояснительной записки:

- Содержание
- Введение
- Сортировка слиянием – итеративный подход и рекурсивный подход
- Быстрая сортировка – итеративный и рекурсивный подход
- Тестирование
- Исходный код
- Использованные источники

Дата выдачи задания: 11.10.2019

Дата сдачи реферата:

Дата защиты реферата: 22.10.2019

Студент

Птухов Д.А.

Преподаватель

Фирсов М.А.

АННОТАЦИЯ

В данной работе была создана программа на языке программирования C++, которая сочетает в себе несколько функций: ввода/вывода массива и его сортировки. Были использованы преимущества C++ для минимизации кода. Для лучшего понимания кода было в нем приведено большое кол-во комментариев и отладочных выводов. Также была проведена его оптимизация с целью экономии выделяемой в процессе работы памяти и улучшения быстродействия программы.

SUMMARY

In this work, a program was created in the C ++ programming language, which combines several functions: input / output of an array and its sorting. The benefits of C ++ were used to minimize code. For a better understanding of the code, a large number of comments and debugging conclusions were given in it. Also, its optimization was carried out in order to save the memory allocated in the process of working and improve the speed of the program.

СОДЕРЖАНИЕ

Введение.....	5
1. Сортировка слиянием	6
Итеративный подход.....	6
Рекурсивный подход.....	6
Итог	6
2. Быстрая сортировка	7
Итеративный подход.....	7
Рекурсивный подход.....	7
Итог	8
3. Функции и структуры данных	9
4. Интерфейс программы	10
5. Тестирование	11

Введение

Целью данной курсовой работы является реализация быстрой сортировки и сортировки слиянием итеративным и рекурсивным методом. Сортировки применяются во многих областях, поэтому создание эффективной сортировки является приоритетной целью. Существует много видов сортировок, однако лишь малая часть из них сможет сравниться по скорости с вышеперечисленными. Каждая из них имеет свои достоинства и недостатки, однако, на дистанции быстрая сортировка опережает сортировку слиянием, поэтому она и является основной.

1. СОРТИРОВКА СЛИЯНИЕМ

1.1. Итеративный подход

Для решения поставленной подзадачи была реализована шаблонная функция `mergeItSort`, которая принимает сортируемый массив и функцию, необходимую для сравнения данных в полученном массиве. Далее при помощи двойного цикла осуществляется последовательная сортировка половинных блоков и их дальнейшее слияние в один отсортированный блок. Далее данные из отсортированного блока переписываются обратно в массив, и вышеописанный алгоритм повторяется, пока не будет отсортирован весь массив. Сложность данного алгоритма постоянна и равна $O(n * \log(n))$, где n – размер исходного массива.

1.2. Рекурсивный подход

Для решения поставленной подзадачи была реализована шаблонная функция `mergeRecSort`, которая принимает сортируемый массив и функцию, необходимую для сравнения данных в полученном массиве. Далее входной массив делится на 2 равных блока и осуществляется рекурсивный вызов исходной функции от ранее сформированных блоков. Для массива единичной длины осуществляется выход из рекурсии. Далее происходит слияние двух ранее отсортированных половинных блоков в один. Сложность данного алгоритма постоянна и равна $O(n * \log(n))$, где n – размер исходного массива.

Итог:

Каждый из видов сортировки имеет свои достоинства и недостатки, так у рекурсивной реализации существует вероятность переполнения стека, что, несомненно, является ее существенным недостатком, однако, с другой стороны рекурсивная реализация имеет понятный и минимизированный код, что является ее большим преимуществом. Однако основными параметрами сортировки являются ее скорость и расходы по памяти, в случае рекурсивной и итеративной реализации сортировки слиянием они равносильны.

2. БЫСТРАЯ СОРТИРОВКА

2.1. Итеративный подход

Для решения поставленной подзадачи была реализована шаблонная функция `quickRecSort`, которая принимает сортируемый массив и функцию, необходимую для сравнения данных в полученном массиве. Далее при помощи циклов и стека, в который сохраняются очередные рассматриваемые границы, осуществляется последовательная перестановка элементов, стоящих до опорного и больших его, с элементами, стоящими после опорного и меньших его. Выход из цикла осуществляется посредством проверки стека на пустоту. Сложность алгоритма варьируется от $O(n * \log(n))$ до $O(n * n)$. Она зависит от выбранного опорного эл-та. Так при выборе максимально/минимального значения массива длина исходного массива сокращается на 1 элемент. Если аналогичный выбор произойдет в остальных случаях, то скорость алгоритма упадет до $O(n*n)$. Затраты по памяти равны 0.

2.2. Рекурсивный подход

Для решения поставленной подзадачи была реализована шаблонная функция `quickRecSort`, которая принимает сортируемый массив и функцию, необходимую для сравнения данных в полученном массиве. Далее в вышеописанной функции создается 3 массива `smaller`, `bigger`, `equal`, которые содержат элементы меньшие, большие и равные ранее выбранному опорному элементу. Далее данные массивы заполняются и осуществляется рекурсивный вызов исходной функции от массивов `smaller` и `bigger`. Пользуясь тем, что массив длиной 1 считается отсортированным, был реализован выход из рекурсии. Далее входной массив заменяется на сумму массивов `smaller`, `bigger`, `equal` при помощи перегрузки оператора `+` для массивов. Сложность алгоритма варьируется от $O(n * \log(n))$ до $O(n * n)$. Затраты по памяти равны $n * \log(n)$.

Итог:

Скорость работы обеих реализация идентична, однако, рекурсивный метод сильно проигрывает по памяти, а также существует вероятность переполнения стека, что делает эту реализацию менее эффективной.

3. ФУНКЦИИ И СТРУКТУРЫ ДАННЫХ

Для считывания данных из входного файла была реализована функция `readFileData`, которая при помощи конструкции `while (std::getline(in, currentFileString))`, осуществляет запись строк из входного потока `in` в строчный массив, который был ранее принят по ссылке.

Для определения типа данных хранящихся в массиве был реализована функция `determineType`, которая осуществляет последовательное преобразование входной строки в другие типы. В случае успешного определения типа функция возвращает его код, в противном случае тип данных остается строчным.

Далее в зависимости от ранее определенного типа осуществляется формирование массива, которое осуществляется при помощи функции `formArr`, она, в свою очередь, при помощи последовательного перебора эл-ов массива во входной строке формирует массив.

Далее осуществляется сортировка массива любым из 4 методов и запись его в выходной файл заранее переданным пользователем.

4. ИНТЕРФЕЙС ПРОГРАММЫ

1. Ввод – файловый, пути до файлов передаются как аргументы командной строки.
2. Выбор предоставляемый пользователю в ходе программы представлен на рисунке 1.

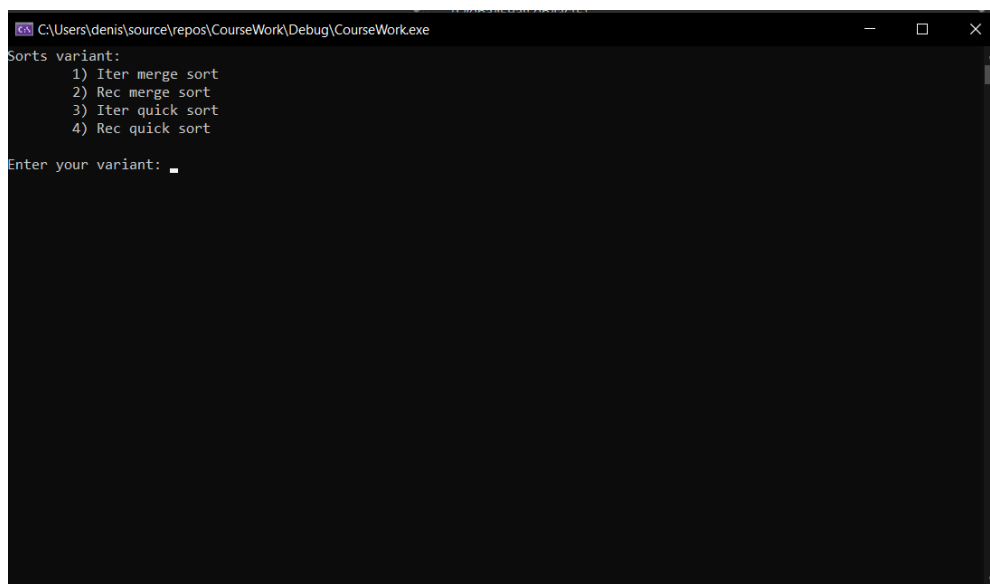


Рисунок 1 – Диалоговое окно выбора сортировки

3. Отладочные выводы, полученные в ходе выполнения, представлены на рисунке 2.

```
Entered array:
5 4 3 2 1

Formed bigger, equal and smaller arrays for stage №0:
5 4 3 2 1
Big:
5 4
Equal:
3
Small:
2 1
Formed bigger, equal and smaller arrays for stage №1:
2 1
Big:
2
Equal:
1
Small:
Empty
Formed bigger, equal and smaller arrays for stage №2:
5 4
Big:
5
Equal:
4
Small:
Empty
Sorted array:
1 2 3 4 5
```

Рисунок 2 –Отладочные выводы программы

5. ТЕСТИРОВАНИЕ

INPUT	OUTPUT
5 4 -10 7 8 15 19 100 8888 1 2	-10 1 2 4 5 7 8 15 19 100 8888
1	1
5 4 3 2 1	1 2 3 4 5
8 7 -10 16 38 13 19 5 -1	-10 -1 5 7 8 13 16 19 38
1 2 3 4 5	1 2 3 4 5
0 0 0 0 0 0	0 0 0 0 0 0
8.1 8.01 8.02 9.03 8.001 9.02 9.00001	8.001 8.01 8.02 8.1 9.00001 9.02 9.03
17 27 0 7 40	0 7 17 27 40
A 17 H AKDL	Array elements don't have same type
KKKK AAA A YH GHOSN	A AAA GHOSN KKKK YH
NDNG DKNbn JBG WNNJK GJNOANJ	DKNbn GJNOANJ JBG NDNG WNNJK
A B C D E	A B C D E
EEE EE E EEEE EEEEE	E EE EEE EEEE EEEEE
YO YOY YOYO YOYOY YOOOOOOOOOOOOOOO	YO YOOOOOOOOOOOOOOO YOY YOYO YOYOY
()((((O)))) (((((((O))) ()))(

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы была написана программа, содержащая в себе итеративную и рекурсивную реализацию быстрой сортировки и сортировки слиянием. Был получен опыт работы с дополнительными возможностями C++ и эффективной алгоритмизацией на нем. Также были закреплены знания полученные на протяжении семестра. Исходный код программы находится в приложении А.

ПРИЛОЖЕНИЕ А

СОДЕРЖИМОЕ ФАЙЛА COURSEWORK.CPP

```
#include "Header.h"

ReturnType streamsCheck(std::ifstream& in, std::ofstream& out)
{
    return (in && out) ? ReturnType::Correct : ReturnType::IncorrectStreams;
}

TypeCode determineType(std::string const& checkString)
{
    TransformPair<int> intTransform = from_string<int>(checkString);
    if (intTransform.transformResult == true)
        return TypeCode::TypeInt;

    TransformPair<char> charTransform = from_string<char>(checkString);
    if (charTransform.transformResult == true)
        return TypeCode::TypeChar;

    TransformPair<double> doubleTransform = from_string<double>(checkString);
    if (doubleTransform.transformResult == true)
        return TypeCode::TypeDouble;

    return TypeCode::TypeString;
}

void readFileData(std::ifstream& in, StringVector& fileData)
{
    std::string currentFileString;

    while (std::getline(in, currentFileString))
    {
        if (currentFileString.back() == '\\r')
            currentFileString.erase(currentFileString.end() - 1);

        fileData.push_back(currentFileString);
    }
}

int main(int argc, char** argv)
{
    srand(time(0));

    if (argc > 2)
    {
        std::ifstream in(argv[1]);
        std::ofstream out(argv[2]);
        ReturnType streamsCheckResult = ReturnType::Correct;

        streamsCheckResult = streamsCheck(in, out);
        if (streamsCheckResult == ReturnType::IncorrectStreams)
        {
            out << "Incorrect streams\\n";
            return 0;
        }

        StringVector fileData;
        readFileData(in, fileData);

        for (auto it = fileData.begin(); it != fileData.end(); ++it)
```

```

{
    std::string& arrStringForm = *it;

    auto searchResult = std::find(arrStringForm.begin(),
arrStringForm.end(), ' ');
    if (searchResult == arrStringForm.end())
    {
        out << arrStringForm << '\n';
        continue;
    }

    std::string firstElement(arrStringForm.begin(), searchResult);
    TypeCode type = determineType(firstElement);
    auto cmp = [](auto a, auto b) {return a < b; };
    auto reverseCmp = [](auto a, auto b) {return a > b; };

    switch (type)
    {
    case TypeCode::TypeInt:
    {
        std::vector<int> arr;
        ReturnType formResult = formArr(arr, arrStringForm);
        if (formResult == ReturnType::IncorrectFileData)
        {
            out << "Array elements don't have same type\n";
            continue;
        }

        quickRecSort(arr, cmp);
        for (auto i : arr)
            out << i << ' ';
        out << '\n';

        break;
    }
    case TypeCode::TypeChar:
    {
        std::vector<char> arr;
        ReturnType formResult = formArr(arr, arrStringForm);
        if (formResult == ReturnType::IncorrectFileData)
        {
            out << "Array elements don't have same type\n";
            continue;
        }

        quickRecSort(arr, cmp);
        for (auto i : arr)
            out << i << ' ';
        out << '\n';

        break;
    }
    case TypeCode::TypeDouble:
    {
        std::vector<double> arr;
        ReturnType formResult = formArr(arr, arrStringForm);
        if (formResult == ReturnType::IncorrectFileData)
        {
            out << "Array elements don't have same type\n";
            continue;
        }

        quickRecSort(arr, cmp);
    }
}

```

```

        for (auto i : arr)
            out << i << ' ';
        out << '\n';

        break;
    }
    case TypeCode::TypeString:
    {
        std::vector<std::string> arr;
        ReturnTypе formResult = formArr(arr, arrStringForm);
        if (formResult == ReturnTypе::IncorrectFileData)
        {
            out << "Array elements don't have same type\n";
            continue;
        }

        quickRecSort(arr, cmp);
        for (auto i : arr)
            out << i << ' ';
        out << '\n';

        break;
    }
    default:
        out << "Incorrect type!\n";
        continue;
    }
}

return 0;
}

```

СОДЕРЖИМОЕ ФАЙЛА HEADER.H

```

#pragma once

#include "SortsHeader.h"
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>

using namespace sorts;
using StringVector = std::vector<std::string>;

enum class ReturnTypе
{
    IncorrectStreams,
    IncorrectType,
    IncorrectFileData,
    Correct
};

enum class TypeCode
{
    TypeInt,
    TypeChar,
    TypeDouble,
    TypeString
};

```

```

template<typename T>
struct TransformPair
{
    TransformPair(T newVal, bool newTransformResult) : value(newVal),
transformResult(newTransformResult)
    { }

    T value;
    bool transformResult;
};

void readFileData(std::ifstream&, StringVector&);
ReturnType streamsCheck(std::ifstream&, std::ofstream&);
TypeCode determineType(std::string const&);

template <typename T>
TransformPair<T> from_string(std::string const& checkString)
{
    T value;
    std::istringstream stream(checkString);
    stream >> value;

    if (stream.fail() || stream.peek() != EOF)
        return TransformPair<T>(value, false);

    return TransformPair<T>(value, true);
}

template <typename T>
ReturnType reformArr(std::vector<T>& arr, std::string const& arrStringForm)
{
    auto startPosition = arrStringForm.begin();

    while (1)
    {
        auto searchResult = std::find(startPosition, arrStringForm.end(), '
');
        if (searchResult == arrStringForm.end())
        {
            TransformPair<T> transformToT =
from_string<T>(std::string(startPosition, arrStringForm.end()));
            if (transformToT.transformResult == false)
                return ReturnType::IncorrectFileData;

            arr.push_back(transformToT.value);
            return ReturnType::Correct;
        }
        TransformPair<T> transformToT =
from_string<T>(std::string(startPosition, searchResult));
        if (transformToT.transformResult == false)
            return ReturnType::IncorrectFileData;

        arr.push_back(transformToT.value);

        startPosition = searchResult + 1;
    }

    return ReturnType::Correct;
}

template <typename T>
ReturnType formArr(std::vector<T>& arr, std::string const& arrStringForm)
{

```



```

        ReturnTypе reformingResult = reformArr(arr, arrStringForm);
        if (reformingResult == ReturnTypе::IncorrectFileData)
            return ReturnTypе::IncorrectFileData;

        return ReturnTypе::Correct;
    }

    template <typename T, typename FUNC_T>
    void sortArr(std::vector<T>& arr, FUNC_T const& cmp)
    {
        quickItSort(arr, cmp);
    }

```

СОДЕРЖИМОЕ ФАЙЛА SOURCEHEADER.H

```

#pragma once
#include <vector>
#include <stack>
#include <algorithm>
#include <time.h>
#include <memory>

namespace sorts
{
    template <typename T>
    std::vector<T> operator+(std::vector<T> const& left, std::vector<T> const&
right)
    {
        std::vector<T> result(left);
        for (auto i : right)
            result.push_back(i);

        return result;
    }

    template<typename T, typename FUNC_T>
    void quickRecSort(std::vector<T>& arr, FUNC_T const& cmp)
    {
        if (arr.size() <= 1)
            return;

        std::vector<T> smaller;
        std::vector<T> bigger;
        std::vector<T> equal;

        size_t supportIndex = rand() % arr.size();
        T supportingElement = arr[supportIndex];

        for (auto i : arr)
        {
            if (i == supportingElement)
                equal.push_back(i);
            else if (cmp(supportingElement, i))
                bigger.push_back(i);
            else
                smaller.push_back(i);
        }

        quickRecSort(smaller, cmp);
        quickRecSort(bigger, cmp);

        arr = smaller + equal + bigger;
    }

```

```

    }

    template<typename T, typename FUNC_T>
    void mergeRecSort(std::vector<T>& arr, FUNC_T const& cmp)
    {
        if (arr.size() <= 1)
            return;

        std::vector<T> leftPart(arr.begin(), arr.begin() + arr.size() / 2);
        std::vector<T> rightPart(arr.begin() + arr.size() / 2, arr.end());

        mergeRecSort(leftPart, cmp);
        mergeRecSort(rightPart, cmp);

        std::vector<T> result;
        size_t leftArrIndex = 0;
        size_t rightArrIndex = 0;

        while (leftArrIndex < leftPart.size() && rightArrIndex <
rightPart.size())
        {
            if (cmp(leftPart[leftArrIndex], rightPart[rightArrIndex]))
            {
                result.push_back(leftPart[leftArrIndex]);
                leftArrIndex++;
            }
            else
            {
                result.push_back(rightPart[rightArrIndex]);
                rightArrIndex++;
            }
        }

        while (leftArrIndex < leftPart.size())
        {
            result.push_back(leftPart[leftArrIndex]);
            leftArrIndex++;
        }

        while (rightArrIndex < rightPart.size())
        {
            result.push_back(rightPart[rightArrIndex]);
            rightArrIndex++;
        }

        arr = std::move(result);
    }

    template<typename T, typename FUNC_T>
    void mergeItSort(std::vector<T>& arr, FUNC_T const& cmp)
    {
        for (size_t currentttBlockSize = 1; currentttBlockSize < arr.size();
currentttBlockSize *= 2)
        {
            for (size_t blockBorder = 0; blockBorder < arr.size() -
currentttBlockSize; blockBorder += 2 * currentttBlockSize)
            {
                size_t leftBorder = blockBorder;
                size_t middleBorder = leftBorder + currentttBlockSize;
                size_t rightBorder = (middleBorder + currentttBlockSize <
arr.size()) ? middleBorder + currentttBlockSize : arr.size();
                std::vector<T> sortedBlock;

```

```

        size_t leftArrIndex = 0;
        size_t rightArrIndex = 0;
        while (leftBorder + leftArrIndex < middleBorder &&
middleBorder + rightArrIndex < rightBorder)
        {
            T currentLeftArrElement = arr[leftBorder +
leftArrIndex];
            T currentRightArrElement = arr[middleBorder +
rightArrIndex];

            if (cmp(currentLeftArrElement,
currentRightArrElement))
            {
                sortedBlock.push_back(currentLeftArrElement);
                leftArrIndex++;
            }
            else
            {
                sortedBlock.push_back(currentRightArrElement);
                rightArrIndex++;
            }
        }

        while (leftBorder + leftArrIndex < middleBorder)
        {
            sortedBlock.push_back(arr[leftBorder +
leftArrIndex]);
            leftArrIndex++;
        }

        while (middleBorder + rightArrIndex < rightBorder)
        {
            sortedBlock.push_back(arr[middleBorder +
rightArrIndex]);
            rightArrIndex++;
        }

        for (size_t insertIndex = leftBorder; insertIndex <
rightBorder; insertIndex++)
        {
            arr[insertIndex] = sortedBlock[insertIndex -
leftBorder];
        }
    }
}

template<typename T, typename FUNC_T>
void quickItSort(std::vector<T>& arr, FUNC_T const& cmp)
{
    std::stack<int> indexStack;
    indexStack.push(arr.size() - 1);
    indexStack.push(0);

    int leftBorder = 0;
    int rightBorder = 0;
    int leftIndex = 0;
    int rightIndex = 0;

    do
    {

```

```

        leftBorder = indexStack.top();
        indexStack.pop();
        rightBorder = indexStack.top();
        indexStack.pop();

        if (rightBorder - leftBorder == 1 && arr[leftBorder] >
arr[rightBorder])
        {
            std::swap(arr[leftBorder], arr[rightBorder]);
        }
        else
        {
            size_t supportIndex = rand() % arr.size();
            T supportingElement = arr[supportIndex];

            leftIndex = leftBorder;
            rightIndex = rightBorder;
            do
            {
                while (supportingElement > arr[leftIndex])
                    leftIndex++;

                while (arr[rightIndex] > supportingElement)
                    rightIndex--;

                if (leftIndex <= rightIndex)
                {
                    std::swap(arr[leftIndex], arr[rightIndex]);
                    leftIndex++;
                    rightIndex--;
                }

            } while (leftIndex <= rightIndex);
        }

        if (leftBorder < rightIndex)
        {
            indexStack.push(rightIndex);
            indexStack.push(leftBorder);
        }

        if (leftIndex < rightBorder)
        {
            indexStack.push(rightBorder);
            indexStack.push(leftIndex);
        }

    } while (!indexStack.empty());
}

```